Syntax-Directed Amorphous Slicing

Mark Harman	**Malcolm Munro**	**Dave Binkley**	**Sebastian Danicic**
Lin Hu	**Xingyuan Zhang**	Loyola College	**Mohammed Daoudi**
Brunel University	University of Durham	4501 North Charles Street	**Lahcen Ouarbya**
Uxbridge, Middlesex	South Road, Durham	Baltimore,	Goldsmiths College
UB8 3PH, UK.	DH1 3LE, UK.	MD 21210-2699, USA.	New Cross, London
			SE14 6NW, UK.

Keywords: Amorphous Slicing, Conditioned slicing, Transformation, WSL, FermaT

Abstract

An amorphous slice of a program is constructed with respect to a set of variables. The amorphous slice is an executable program which preserves the behaviour of the original on the variables of interest. Unlike syntax–preserving slices, amorphous slices need not preserve a projection of the syntax of a program. This makes the task of amorphous slice construction harder, but it also often makes the result thinner and thereby preferable in applications where syntax preservation is unimportant.

This paper describes an approach to the construction of amorphous slices which is based on the Abstract Syntax Tree of the program to be sliced, and does not require the construction of control flow graphs nor of program dependence graphs. The approach has some strengths and weaknesses which the paper discusses.

The amorphous slicer, is part of the GUSTT slicing system, which includes syntax preserving static and conditioned slicers, a side effect removal transformation phase, slicing criterion guidance and for which much of the correctness proofs for transformation steps are mechanically verified. The system handles a subset of WSL, into which more general WSL constructs can be transformed.

The paper focuses upon the way in which the GUSTT System uses dependence reduction transformation tactics. Such dependence reduction is at the heart of all approaches to amorphous slicing. The algorithms used are described and their performance is assessed with a simple empirical study of best and worst case execution times for an implementation built on top of the FermaT transformation system for maintenance and re–engineering.

1 Introduction

Program slicing is an automated source code extraction technique which produces a version of a program that preserves a projection of the original program's semantics [10, 24, 40, 71, 78]. Traditionally, this projection is defined in terms of a subset of variables of interest and is constructed using the sole transformation of statement deletion [78]. The slice is therefore a subprogram which preserves a subcomputation.

Amorphous slicing [9, 37, 43] is a variation of traditional slicing, in which the semantic property is retained (the slice maintains a subcomputation), while the syntactic restriction to statement deletion is relaxed. By sacrificing the connection with syntax, smaller slices can be produced. Amorphous slicing is therefore useful in applications of slicing where syntax is less important than size reduction. For example, in applications of slicing to re–engineering where the program is, by definition, to be syntactically altered, amorphous slicing may be more suitable than syntax–preserving slicing. However, for applications like debugging, where syntax is important, amorphous slicing is less likely to be suitable compared to syntax–preserving slicing.

Transformations which tend to reduce program size are like slices in that they preserve the (projected) semantics of the program, but are unlike slicing because they do not preserve syntax. On the other hand, slicing is like transformation in the way it preserves the semantics of the program for some projection of the original semantics. When slicing and transformation are combined, the result is *amorphous slicing.*

According to the definition of Amorphous Slicing, an amorphous slice can be constructed using any program transformation which simplifies the program and which preserves the effect of the original program with respect to the slicing criterion. This syntactic freedom allows amorphous slicing to perform greater simplification with

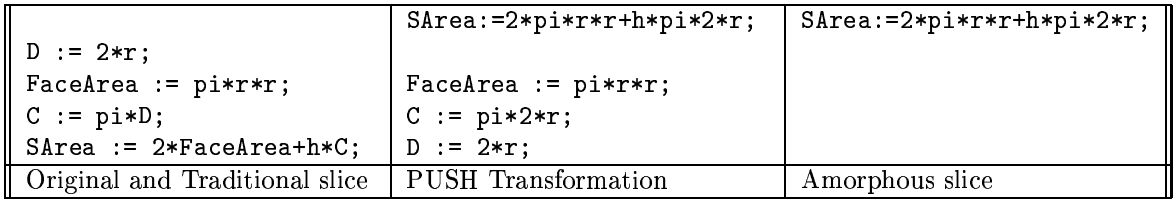


Figure 1: Amorphous Slicing Produces Thinner Slices by Removing Syntactic Restrictions

the result that amorphous slices need never be larger than their syntax-preserving counterparts. Often they are considerably smaller.

Amorphous slicing can be thought of as a generalisation of slicing: the semantic requirement is retained, but the syntactic requirement is relaxed. This makes amorphous slicing more like transformation than traditional slicing. Similarly, amorphous slicing can be thought of as a generalization of transformation: the semantic requirement to preserve the meaning of the program is relaxed to the requirement that the (projected) semantics *with respect to the slicing criterion* is preserved.

Amorphous slicing allows a slicing tool to 'wring out' the semantics of interest, using transformation to reduce the dependencies in the original program and then using traditional syntax preserving slicing to extract the components of the computation which have been isolated by transformation. Consider the simple program fragment in the leftmost section of Figure 1. This program computes face and surface areas for a cylinder of radius `r` and height `h`. The computations of each value are built up in terms of previous values creating dependencies between the statements. This inter-dependence inhibits the simplification abilities of traditional, syntax-preserving slicing. For example, the syntax-preserving slice on the final value of the variable `SArea` contains the whole program.

Consider the version of the original program in the central section of Figure 1. This version of the program has been produced using a simple code motion transformation (Rule 2 of Figure 7). The rule pushes an assignment forward in the program, re-writing the expressions past which the assignment moves. The effect of this transformation is meaning preserving, but it breaks the interdependence which inhibits traditional slicing. For example, in the example in Figure 1, the variable `SArea` depends upon all four assignment statements; in the 'push transformed' version it depends on only one.

Dependence reduction is not the only way in which transformation can be used to improve slicing, it is simply one of the most obvious ways in which transformation can improve slicing, once the requirement of syntax-preservation is removed. For example, the amorphous slice on the variable `SArea` in Figure 1 could be improved by expression simplification, which could replace the expression

```
SArea:=2*pi*r*r+h*pi*2*r;
```

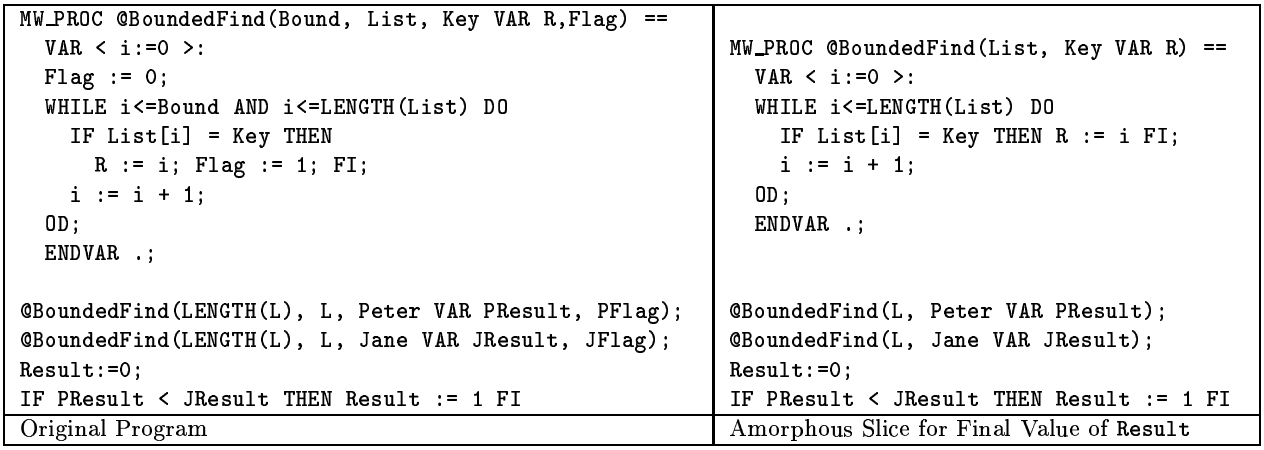
with the expression:

```
SArea:=2*pi*r*(r+h);
```

This paper describes the implementation of an amorphous slicing tool that mixes domain independent dependence reduction and simplification transformations with domain specific transformations, pre-processing transformation and traditional slicing to construct amorphous slices. The paper describes the architecture of the overall system and the dependence reduction transformation component in detail, and illustrates how each phase combines to produce simpler slices using a worked example.There is an implementation of the amorphous slicer available for experimentation via web-interface at `http://www.brunel.ac.uk/~cssrllh/slicer/`.

Some of the transformation tactics used by the GUSTT system may also find application in other transformation programs. For example the side effect removal tactic removes intraprocedural side effects, while the dependence reduction tactic reduces the inter-dependence between statements. The former may be useful as an enabling step to other transformation-based approaches [5, 12, 59, 74], while the latter may be useful in work on transformation-based automatic parallelisation [55, 58, 66, 79].

The rest of this paper is organised as follows. Section 2 explains why amorphous slice construction is a harder problem than syntax-preserving slice construction. Sections 3 and 4 describe the overall GUSTT amorphous



```
MW_PROC @BoundedFind(Bound, List, Key VAR R,Flag) ==
  VAR < i:=0 >:
  Flag := 0;
  WHILE i<=Bound AND i<=LENGTH(List) DO
    IF List[i] = Key THEN
      R := i; Flag := 1; FI;
    i := i + 1;
  OD;
  ENDVAR .;

@BoundedFind(LENGTH(L), L, Peter VAR PResult, PFlag);
@BoundedFind(LENGTH(L), L, Jane VAR JResult, JFlag);
Result:=0;
IF PResult < JResult THEN Result := 1 FI
```

Original Program

```
MW_PROC @BoundedFind(List, Key VAR R) ==
  VAR < i:=0 >:
  WHILE i<=LENGTH(List) DO
    IF List[i] = Key THEN R := i FI;
    i := i + 1;
  OD;
  ENDVAR .;


@BoundedFind(L, Peter VAR PResult);
@BoundedFind(L, Jane VAR JResult);
Result:=0;
IF PResult < JResult THEN Result := 1 FI
```

Amorphous Slice for Final Value of `Result`

Figure 2: A Program Fragment and one of its Interprocedural Amorphous Slices

slicing system, and the details of the algorithms used for Dependence Reduction Transformation. Section 5 presents a simple worked example which illustrates the dependence reduction transformation process. Section 6 compares the abstract-syntax tree approach described here with the more conventional approach based upon the System Dependence Graph (SDG), highlighting strengths and weaknesses of each approach. Section 7 gives the results of a simple empirical study of the best and worst case performance times for LinIAS, an implementation of the Amorphous Slicer for WSL. Section 8 shows how amorphous static slicing can be combined with syntax-preserving conditioned slicing to produce amorphous conditioned slices. Section 9 presents a brief overview of related work. Sections 11 and 10 present conclusions and future work.

This paper is an extension of a paper presented at the 2^{nd} IEEE workshop on Source Code Analysis and Manipulation (SCAM 2002). Some of the material presented here has previously appeared: The example in Section 5 has been adapted from a paper [43] presented at the 1^{st} IEEE Workshop on Analysis, Slicing, and Transformation (AST 2001). Some of the empirical results in Section 7 were presented at the SCAM 2002 workshop.

2 Amorphous Slicing is Harder than Syntax–Preserving Slicing

The construction of amorphous slices is harder than the construction of syntax-preserving slices, because *any* transformation can potentially be applied in amorphous slicing. Previously published algorithms for amorphous slicing [9, 43] have concerned only intraprocedural transformations. This paper presents a simple interprocedural amorphous slicing algorithm and initial results on the performance of the dependence reduction transformation at the heart of our amorphous slicing algorithm. The current approach 'pushes' the intraprocedural analysis into procedure and function bodies. However, even given the limitations of the current approach, it is possible to construct some interesting slices which illustrate the way in which interprocedural amorphous slicing improves upon interprocedural syntax-preserving slicing.

For example, consider the example in the leftmost section of Figure 2. The program is written in WSL. The syntax of the subset of WSL used in this paper is presented in Figure 3. The main program consists of the final four lines of code. The main program makes two calls to the procedure called `@BoundedFind` (the presence of the `@` symbol is required by WSL, but it is unimportant). The first three parameters to this function are `Bound`, `List` and `Key` and they are all passed by value. The remaining two parameters, `R` and `Flag` are passed by value-result, and constitute the results of the procedure. The `VAR ... ENDVAR` construct introduces a local variable, `i`, which is used as a `WHILE` loop counter. The rest of the program uses the Algol-like subset of WSL.

The procedure `@BoundedFind` searches the list passed to it, until either an upper bound, specified by the formal parameter `Bound`, is reached or the end of the list is reached. If the element `Key` is located by this search, then `R` is set to the index of the element and `Flag` is set to 1 (which denotes true). Otherwise, `R` is unaffected and `Flag` is set to 0 (which denotes false).

Assignment statement:

$$\text{x} := \text{e}$$

Conditional statement:

$$\text{IF } B \text{ THEN } S_1 \text{ ELSE } S_2 \text{ FI}$$

While statement:

$$\text{WHILE } B \text{ DO } S \text{ OD}$$

Sequence of statements:

$$S_1;\ S_2$$

Procedure:

$$\text{MW\_PROC } @P(X \ \ \text{VAR } Y) == S\ .;$$

where the variable list X is a value parameter list, Y is a value-result parameter list, and S is the body of procedure $@P$.

Procedure call:

$$@P(X \ \ \text{VAR } Y)$$

Function:

$$\text{MW\_FUNCT } @F(X) == \\ \text{VAR} < Y >: S\ (e).;$$

where the variable list X is a value-result parameter list, Y is the local variable list, S is the body, and the function $@F$ returns the value of the expression e.

Function call:

$$@F(X)$$

Figure 3: Interprocedural Subset of WSL

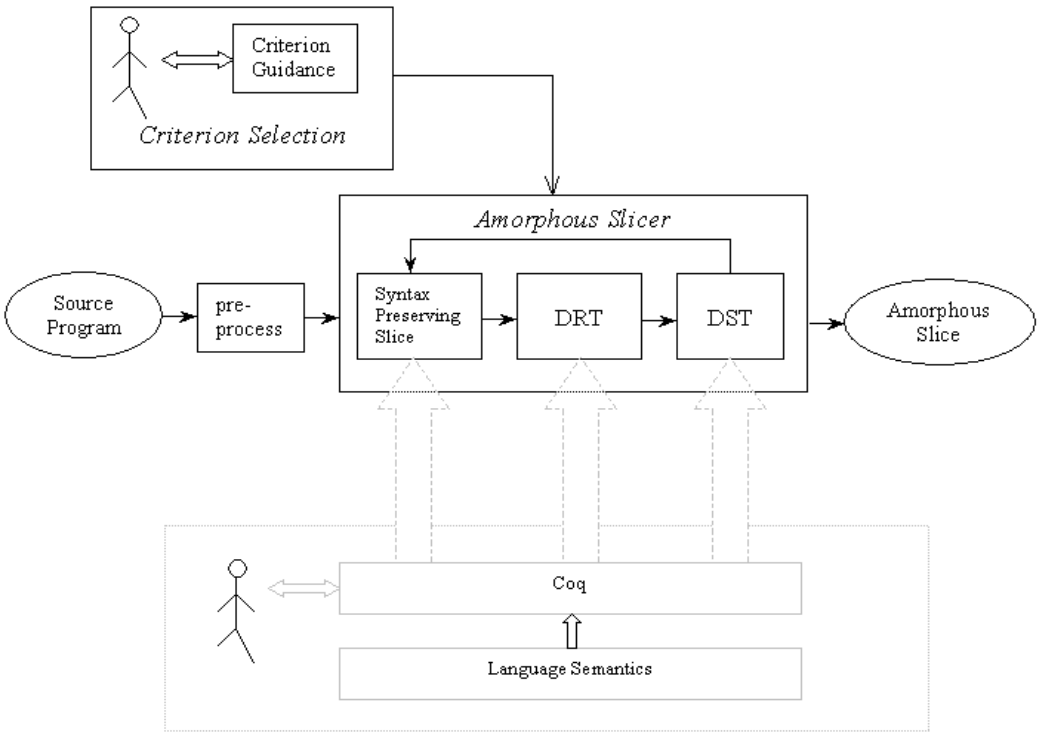


Figure 4: The Architecture of the Overall GUSTT System

The main program attempts to use two calls to `@BoundedFind` to determine whether the element `Peter` occurs strictly before the element `Jane` in the list `L`. It sets the flag variable `Result` accordingly. Slicing on `Result` illustrates one difference between syntax-preserving slicing and amorphous slicing, showing how amorphous slicing allows for more simplification. Observe that, for the purpose of determining the value of `Result`, only the returned parameter `R` is required. The `Flag` parameter can be discarded. Syntax preserving slicing can reveal this, because the slice of the body of the procedure does not contain assignments to the formal parameter `Flag`. However, the parameter `Bound` also is not required in the slice, because the `WHILE` loop test becomes (by substitution of formal for actual parameters and simple code motion):

```
i<=LENGTH(L) AND i<=LENGTH(L)
```

in both calls and this can be simplified to:

```
i<=LENGTH(L)
```

Such a substitution and simplification, makes the formal parameter `Bound` redundant and so it is dropped from the definition and call in the amorphous slice. Syntax-preserving slicing is prevented from exploiting this simplification opportunity, because the syntactic restriction prevents it from substituting an actual parameter for the corresponding formal parameter. The amorphous slice is therefore simpler than the corresponding syntax-preserving slice (although it is not a syntactic subset of it).

3 The Overall System

Figure 4 describes the architecture of the amorphous slicing system developed as part of GUSTT project. The current implementation is available for experiment, via web-form interface at:

`http://www.brunel.ac.uk/~cssrllh/slicer/`.

The GUSTT system consists of a number of components, some of these have been described in detail elsewhere and are only summarised here. These phases are the side effect removal phase [42, 44], the approach to criterion guidance [39] and the approach to formalisation and correctness proof of the transformations used [80, 81]. The purpose of this section is to provide a context for the description of the Dependence Reduction Transformation (DRT) step, which is at the heart of the slicer and which is described in more detail in the next section.

3.1 Criterion Guidance

At present, the criteria selection component is entirely human based. The ultimate goal of the GUSTT research project is to partly automate the criterion selection component of the system to produce guidance to the human in selecting good slicing criteria.

There is a problem with guiding the choice of criterion; the slicing criterion is typically at too low a level for the human maintenance engineer. The maintenance engineer typically wants to ask questions of the form:

> "Given an original program, construct the simplest program that:
>
> - performs the same master file update operation,
> - replicates only the defensive programming features of the original,
> - closes down the reactor under the same conditions,
> - has the same effects upon the back-up disk or
> - communicates in the same way with the thermostat controller."

These questions cannot be asked directly using slicing, because it is not clear how to map concepts such as 'file update' and 'defensive programming' onto program variables. However, recent work [39] has shown that by combining slicing with concept assignment [7, 31, 32, 49], it is possible to construct executable 'concept slices' in which the strengths of both slicing and concept assignment are retained. This allows for meta-level guidance in the choice of the sets of slicing criteria needed in order to construct good slices.

In executable concept slicing, the concept assignment phase locates candidate code based on very high level criteria, such as 'file update' and 'defensive programming'. The concept so-defined is not executable, but slicing is used to 'fill in the gaps', resulting in a subprogram which is both executable, yet which is extracted according to a high level criterion. The high level criterion of concept assignment corresponds to a set of lower level criteria for slicing. This approach to the combination of slicing and concept assignment is described in detail elsewhere [39].

3.2 Pre-Processing

The GUSTT system involves the pre-processing of original programs to achieve side-effect removal [42, 44]. The goal is to produce a version of the program written in a form which is essentially 'functional with sequencing'. This eases further transformation, as transformation rules are typically easier to define for a functional language [3, 22, 51, 63].

A side effect is any change in program state that occurs as a by-product of the evaluation of an expression. A side-effect free expression when evaluated, simply returns its value, causing no change in state. Side effects tend to inhibit transformation, and so they are removed in a pre-processing phase of the system. Figure 5 provides several examples of code fragments with side effects and their side-effect free versions.

The Side Effect Removal Transformation (SERT) [42] rewrites a program which may contain side effects into a semantically equivalent program which is guaranteed to be side-effect free. Software engineering techniques such as slicing, program transformation and other program analysis techniques can benefit from SERT. It is one of the enabling steps in GUSTT as well. The SERT algorithm is achieved using the LinSERT system, described in more detail elsewhere [42, 44] and which is available for non-commercial use from

`http://www.brunel.ac.uk/~csstmmh2/linsert`.

Original	Side-Effect Free Version
`a = ++x + x;`	`undefined`
`b = (++x, ++x);`	`b = x+2; x = x+2;`
`c = ++x + ++x;`	`undefined`
`d = ++x && ++x;`	`if (x == -1) { d = 0; x = 0; }` `else { d = x+2!=0; x = x+2; }`
`f = (x=4) + (y=5);`	`f = 9; x = 4; y = 5;`
`g = (x=4) + (x=5);`	`undefined`

Figure 5: Side-effecting Statements with Side-Effect Free Versions

3.3 Language Semantics: Automated Proof Checking

Underlying all the central phases of the amorphous slicer is the approach to correctness assurance. In the GUSTT system this is a partly automated formal proof process. The Coq proof assistant [15, 16, 18, 64] is used to express and check proofs of correctness for the slicing and transformation components of the system. The semantic basis of WSL is Weakest Precondition (WP) and program transformation. In order to automate proofs about slicing and transformation algorithms in WSL the WP semantics of WSL have been formalised within the Coq type-theoretic proof checking assistant [81]. An operational semantics for WSL has also been constructed and an equivalence between the operation and weakest pre-condition semantics has been established [80].

The advantages of using Coq, is that proof of program transformation can be constructed within the Coq system (by hand) and checked automatically (by the Coq system). This removes one possible source of problems, because it is well-known that computer proofs are not often thoroughly checked by humans [26].

More details on the GUSTT approach to formally assured and machine-checked transformations can be found elsewhere [80, 81].

3.4 The Amorphous Slicer Itself

The central amorphous slicer contains three subcomponents. The three components are iteratively applied until no further reduction in amorphous slice size is possible. This 'top level' algorithm employed by the central amorphous slicing system is described in Figure 6. In this (and other) figures detailing algorithms, the notation **let** a **be** b is used for pattern matching against structures, so that the components of these structures can be selected and so that new structures can be constructed. To avoid confusion between notation used to express the algorithm itself and the syntactic constructs being built, Quine's quasi quotes [69] are used to encase concrete syntactic constructions of WSL statements and expressions. That is, in $[\![S]\!]$, S is some WSL statement. Assignment at the algorithm level (as opposed to the WSL assignment syntactic constructor ':=') is denoted $\leftarrow$, and list structures are operated upon with the usual selectors $Head$ and $Tail$ and with the constructor $Append$ for conjoining two lists or an element and a list.

The initial component of the central amorphous slicer, is a traditional syntax-preserving slicer, which uses a traditional interprocedural slicer for syntax–preserving slicing based upon the Abstract Syntax Tree (AST) [20, 62]. We perform an initial syntax preserving slicing step (before the iteration begins) and then perform syntax preserving slicing again at the end of the body of the iteration. This has the same effect, but, anecdotally, is often faster because the initial slicing step is relatively efficient and if it removes a large portion of code it reduces the size of the problem once the iteration commences.

If syntax preserving slicing and domain specific transformations have no effect then s' and p_1 will be identical and the iteration will terminate. Otherwise, there is some possibility of reduction because both syntax–preserving slicing step and the domain specific transformations are so-designed that they have either no effect or reduce the size of the program. This guarantees that the iteration will, eventually, terminate.

The final component of the central amorphous slicer seeks additional transformation rules for domain specific problems. Domain Specific Transformation (DST) exploits knowledge of the application domain in order to improve the chances of achieving slice size reduction. Earlier work [45] has shown considerable improvement in simplification by DST. The DST component is essentially an opportunistic phase of the approach. Different

```
function AmorphousSlicer(p: Program) returns a Program
declare
   p_1, s, s': Programs
   ProcList: List of procedures
   FuncList: List of functions
   Main: Statement Sequence
begin
   p_1 ← SyntaxPreservingSlicer(p)
   while true do
      let [[Main, ProcList, FuncList]] be p_1
      Main' ← DRT(Main)
      ProcList' ← DRT(ProcList)
      FuncList' ← DRT(FuncList)
      let p_1 be [[Main', ProcList', FuncList']]
      s ← SyntaxPreservingSlicer(p_1)
      s' ← DomainSpecificTransformations(s)
      if s' = p_1
      then
         return (s')
      else
         p_1 ← s'
      fi
   od
end
```

Figure 6: Top Level Interprocedural Amorphous Slicing Algorithm

Push Rules:

Rule 1 (Assignment to Assignment: Absorb)

$$\frac{e_3 = \mathbf{SUB}(e_2, i, e_1)}{[\![i := e_1;\ i := e_2]\!] \Rightarrow [\![i := e_3]\!]}$$

Rule 2 (Assignment to Assignment: MoveToRight)

$$\frac{i_1 \neq i_2, i_2 \notin \mathrm{REF}(e_1), e_3 = \mathrm{SUB}(e_2, i_1, e_1)}{[\![i_1 := e_1;\ i_2 := e_2]\!] \Rightarrow [\![i_2 := e_3;\ i_1 := e_1]\!]}$$

Rule 3 (Assignment to If-Then)

$$\frac{e_2' := \mathbf{SUB}(e_2, i, e_1), [\![i{:=}e_1;\ c]\!] \Rightarrow [\![c'\ \ i{:=}e_1;]\!]}{[\![i{:=}e_1;\ \texttt{IF}\ (e_2)\ \texttt{THEN}\ c\ \texttt{FI}]\!] \Rightarrow [\![\texttt{IF}\ (e_2')\ \texttt{THEN}\ c'\ \texttt{FI};\ i{:=}e_1]\!]}$$

Rule 4 (Assignment to If-Then-Else)

$$\frac{e_2' = \mathbf{SUB}(e_2, i, e_1),\ \ [\![i{:=}e_1;\ \ c_1]\!] \Rightarrow [\![c_1'\ \ i{:=}e_1;]\!],\ \ [\![i{:=}e_1;\ \ c_2]\!] \Rightarrow [\![c_2'\ \ i{=}e_1;]\!]}{[\![i{:=}e_1;\ \ \texttt{IF}\ (e_2)\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2\ \texttt{FI}]\!] \Rightarrow [\![\texttt{IF}\ (e_2')\ \texttt{THEN}\ c_1'\ \texttt{ELSE}\ c_2'\ \texttt{FI};\ \ i{:=}e_1]\!]}$$

Rule 5 (Others: Statement to Statement)

$$\frac{\texttt{DEF}(\texttt{st}_1) \cap \texttt{REF}(\texttt{st}_2) = \phi,\ \texttt{REF}(\texttt{st}_1) \cap \texttt{DEF}(\texttt{st}_2) = \phi, \texttt{DEF}(\texttt{st}_1) \cap \texttt{DEF}(\texttt{st}_2) = \phi}{[\![st_1;\ \ st_2]\!] \Rightarrow [\![st_2;\ \ st_1]\!]}$$

Simplify Transformation Rules:

Rule 6 (Expression) *Using WSL symbolic computation library attempt to simplify expression*

Rule 7 IF (e) SKIP ELSE c FI ⇒ IF Not(e) THEN c FI

Rule 8 IF FALSE THEN c_1 ELSE c_2 FI ⇒ c_2

Rule 9 IF FALSE THEN c FI ⇒ SKIP

Figure 7: Rules Applied by the Top Level Algorithm

applications will require different 'transformation rule sets' tailored to the problem in hand. These transformations are called 'targeted transformations' because they are targeted at a particular problem. Domain specific amorphous slicing has also been used to improve evolutionary testing [41].

The middle component of the GUSTT system, the DRT tactic, aims to remove dependencies, thereby improving the likelihood that subsequent slicing and Domain Specific Transformations will be applicable. DRT consists of a set of transformations aimed at reducing statement inter-dependence. This phase of the amorphous slicing engine is 'general purpose' because the reduction of inter-dependence between statements will always tend to reduce the size of slices and all applications require slices to be as small as possible. The specifics of the initial and final phases of the central slicer are described elsewhere [62, 41, 45]. However, the dependence reduction component of the GUSTT system is a useful transformation tactic in its own right and is germane to any amorphous approach to slicing. It is described in more detail in the next section.

4 Dependency Reduction Transformation

The algorithm uses a set of transformation rules to achieve a form of symbolic execution of the program. Each transformation applied is semantics preserving. The transformation rules used are presented in Figure 7. A rule of the form: $\frac{A}{B \Rightarrow C}$ is interpreted as "If A holds then the fragment B can be transformed to the fragment C." The terms $\mathbf{REF}(e)$ and $\mathbf{DEF}(e)$ denote the referenced and defined variables [2] of expression e. The term $\mathbf{SUB}(e_1, i, e_2)$ returns the expression that results from substituting all occurrences of the variable i in the expression e_1, with the expression e_2.

The rules are implemented by the DRT algorithm (Figure 8), which walks a program's AST applying the auxiliary transformation tactic PUSH (Figure 9). The PUSH tactic takes an assignment statement and attempts to push it forward in a code sequence. As the assignment passes statements which reference its defined variable, these passed statements have to be updated to reflect the symbolic effect of the assignment's execution. The pushing forward of an assignment is achieved by applying the Push Assignment Rule shown in Figure 7.

An assignment can get 'stuck' in the pushing process. This happens when an attempt is made to push an assignment which references a variable v past a statement which defines v. In this situation the PUSH tactic has no effect upon the statement sequence through which the assignment is to be pushed.

For example, consider the simple code sequence

```
a:=b+c;
d:=a+1;
x:=d-a;
```

Computing DRT(⟦`a:=b+c; d:=a+1; x:=d-a;`⟧) requires computing PUSH(⟦`a:=b+c;`⟧, ⟦`d:=a+1; x:=d-a;`⟧). The function PUSH, using Rule 2 of Figure 7, first updates the assignment to `d`. After which *DoneList* is ⟦`d:=(b+c)+1;`⟧ and *WorkList* is ⟦`x:=d-a`⟧. The PUSH algorithm's main loop is repeated, which requires pushing ⟦`a:=b+c;`⟧ past ⟦`x:=d-a;`⟧. After doing so, *DoneList* is ⟦`d:=(b+c)+1; x:=d-(b+c);`⟧ and *WorkList* is empty.

The overall effect of this first call to PUSH, has been to push the first assignment `a:=b+c` past the other two assignments, with the consequent effect of replacing the right hand sides of the two assignments with symbolically evaluated expressions. After the first call to PUSH, *WorkList* in DRT is ⟦`d:=(b+c)+1; x:=d-(b+c);`⟧, and *DoneList* is ⟦`a:=b+c;`⟧. A second iteration of the main loop in DRT calls:

PUSH(⟦`d:=(b+c)+1;`⟧, ⟦`x:=d-(b+c);`⟧)

After this call to PUSH, *WorkList* is ⟦`x:=((b+c)+1)-(b+c);`⟧ and *DoneList* is ⟦`d:=(b+c)+1; a:=b+c;`⟧. DRT now calls PUSH for the last time:

```
function DRT(sl: Statement Sequence) returns a Statement Sequence
declare
   cl, WorkList, DoneList, TopList: Statement Sequences
   b: Predicate Expression
   c, s, s1, s2: Statements
   Stuck: Boolean
begin
   WorkList ← sl
   DoneList ← [ ]
   TopList ← [ ]
   while WorkList ≠ [ ] do
      if head(WorkList) is an assignment statement
      then
         (c,cl,Stuck) ← PUSH(head(WorkList),tail(WorkList))
         if not(Stuck)
         then
            WorkList ← cl
            DoneList ← Append(c,DoneList)
         else
            TopList ← Append(TopList,c)
            WorkList ← tail(WorkList)
         fi
      else     Head(WorkList) is not an assignment statement
         if head(WorkList) is an if-then statement
         then
            let ⟦if b then s⟧ be head(WorkList)
            TopList ← Append(TopList,⟦if b then DRT(s)⟧)
         else
            if head(WorkList) is a while statement
            then
               let ⟦while b do s⟧ be head(WorkList)
               TopList ← Append(TopList,⟦while b do DRT(s)⟧
            else
               if head(WorkList) is an if-then-else statement
               then
                  let ⟦if b then s1 else s2⟧ be head(WorkList)
                  TopList ← Append(TopList,⟦if b then DRT(s1) else DRT(s2)⟧)
               else     unrecognized statement encountered: Leave head(WorkList) untransformed
                  TopList ← Append(TopList,head(WorkList))
               fi
            fi
         fi
         WorkList ← tail(WorkList)      move on to consider the next statement in WorkList
      fi
   od
  return (Append(TopList, DoneList))
end
```

Figure 8: Dependence Reduction Transformation algorithm

```
function PUSH(c: Statement, cl: Statement Sequence) returns a triple (Statement, Statement Sequence, Boolean)
declare  WorkList, DoneList: Statement Sequence
         i, i': Identifiers
         e, e': Expressions
         p, c': Statements
begin
   WorkList ← cl
   DoneList ← []
   while WorkList ≠ [] do
      let ⟦i := e;⟧ be c
      if head(WorkList) is an assignment statement
      then
         let ⟦i' := e';⟧ be head(WorkList)
          if  i = i'
          then      Rule 1 applies — assignment c 'vanishes'
              let ⟦c';⟧ be the result of applying Rule 1 to ⟦i := e; i := e'⟧
              return (⟦skip;⟧, Append(DoneList, Append(c',WorkList)), false)
          else
             if Rule 2 applies to ⟦i := e; i' := e'⟧
             then     c pushes past an assignment at head of WorkList
                let ⟦c'; c;⟧ be result of Rule 2 applied to ⟦i := e; i' := e';⟧
                DoneList ← Append(DoneList,c')
                WorkList ← tail(WorkList)
             else     assignment c has 'got stuck' in front of the assignment at head of WorkList
                return (c, Append(DoneList, WorkList), true)
             fi
          fi
      else     head(WorkList) is not an assignment statement
         if head(WorkList) is a conditional
         then     c pushes past a conditional at head of WorkList
            if Rule 3 or 4 apply to ⟦i := e; head(WorkList)⟧
            then
               let ⟦p; c⟧ be result of Rule 3 or 4 applied to ⟦i := e; head(WorkList)⟧
               DoneList ← Append(DoneList,p)
               WorkList ← tail(WorkList)
            else     assignment c has 'got stuck' in front of a conditional at head of WorkList
               return (c, Append(DoneList, WorkList), true)
            fi
         else     head(WorkList) is some other statement
            if Rule 5 applies to ⟦i := e; head(WorkList)⟧
            then
               let ⟦p; c⟧ be result of Rule 5 applied to ⟦i := e; head(WorkList)⟧
               DoneList ← Append(DoneList,p)
               WorkList ← tail(WorkList)
            else     assignment c has 'got stuck' in front of some other statement at head of WorkList
               return (c, Append(DoneList, WorkList), true)
            fi
         fi
      fi
   od
   Reaching this point means c did not 'get stuck', it passed all statements in cl
   return (c, DoneList, false)
end
```

Figure 9: Push algorithm

PUSH([[x:=((b+c)+1)-(b+c);]], [[]])

After this call *WorkList* is empty and *DoneList* is [[x:=((b+c)+1)-(b+c); d:=(b+c)+1; a:=b+c;]]. The execution of DRT is now complete. The resulting transformation of the code has produced:

```
x:=((b+c)+1)-(b+c);
d:=(b+c)+1;
a:=b+c;
```

Notice how the dependence of the assignment to x upon the assignments to a and b has been 'transformed out' in this process. Slicing the new version of the program with respect to the final value of x will now produce the single assignment:

```
x:=((b+c)+1)-(b+c);
```

whereas previously, it would have produced the entire code sequence.

As the example shows, the push transformations, combined with the application of traditional, syntax preserving slicing, produced greater simplification than traditional slicing alone. The overall effect is that expressions become more complex, but statements become less interdependent, so slices tend to be smaller (or at least have fewer nodes) and tend to assign to fewer variables. Of course, the expression assigned to x has increased in size as a result of the transformation process. However, in this case, simple expression simplification improves the final amorphous slice on x to:

```
x:=1;
```

5 A Worked Example

This section illustrates the GUSTT system using the example in Figure 10. The example is presented in C, so that the side effect removal phase can be illustrated. The amorphous slicer is implemented for WSL, which is side effect free, so the example in C, since side effects have been removed, is re-written in WSL. The side effect removal phase, and the amorphous slicing of WSL are implemented and publicly available. The conversion from C to WSL is currently performed by hand.

The example program is a program which is intended to compute the mean value for the elements in the array A and the means for the even and odd numbers of the array. These values are used to compute the difference between the overall mean and the mean for evens and odds. However, the program contains a *bug*; the assignment operator = has been used in place of the equality test operator == in testing a flag variable evenflag which is used to denote the outcome of testing whether or not a particular element of the array is even or not. It has been previously observed [6] that transformation has a tendency to highlight bugs in programs, because the transformation is meaning preserving and yet it renders the program in a different syntax. This worked example shows how the combination of slicing *and* transformation can further improve transformation's bug-revealing tendency.

Of course, no set of transformation rules can ever be complete and so there will always be amorphous slices which exist in theory, but which are not found by any algorithm. Indeed, statement–minimal syntax–preserving slices are not generally computable, so even a complete set of transformations would not guarantee minimal amorphous slices. However, the example shows that even with our limited set of transformation rules, combined with traditional syntax–preserving slicing, it is possible to make progress.

The rest of this section steps through the application of the various components of the GUSTT system, showing how each affects the example. First, the side-effects in program P_0 are removed. The side-effect free

program P_1 (shown in Figure 11) is obtained. Not surprisingly, as the bug involves the mistaken use of a side effect the side-effect removal tactic makes the bug more obvious. That is, the side effect free version of the program contains the tell-tale conditional:

```
if (0)
```

This code appears to be anomalous, because it clearly can only evaluate to `false`, making the `then` branch of the conditional redundant. As will be shown, further transformation and slicing, make this bug even more evident.

```
total=0;  i=0;
evens=0;  noevens=0;
odds=0;   noodds=0;
scanf("%d",&n);
while (i<=n){
  evenflag = A[i] % 2;
  if(evenflag=0)      /* bug */
    {evens = evens + A[i];
     noevens++; }
  else
    {odds = odds + A[i];
     noodds++; }
  total = total + A[i++];
}
if(noevens!=0) meaneven=evens/noevens;
else meaneven = 0;
if(noodds!=0) meanodd = odds/noodds;
else meanodd = 0;
mean = total/++n;
evendifference = abs(meaneven-mean);
odddifference = abs(meanodd-mean);
```

Figure 10: Program P_0: The program to be sliced with respect to `evendifference`

The LinIAS implementation of the GUSTT amorphous slicer takes and produces programs in WSL. Figure 12 presents the WSL version of the side effect free `C` program in Figure 11. Slicing this WSL program, P_2, on the variable `evendifference`, using syntax-preserving slicing produces the conventional slice, S_0, in Figure 13.

Traditional syntax-preserving slicing [48, 71, 78] does not exploit the fact that `if(0)` can never execute the `then` branch (line 9 and 10 in Figure 13). However, Tip [70] and Ernst [27] show how compiler optimization techniques can be used to transform an intermediate representation that would allow optimized syntax preserving slices to be constructed. Such optimizations are included in the DRT rule set when they reduce dependence. For example, Rule 8 and Rule 9 in Figure 7 are standard redundant computation removal optimizations. Applying Rule 9 to the code fragment in Figure 13 causes the code from lines 8 to 10 of S_0 to be deleted. Applying Rule 8, the code line 17 and 18 of S_0 are transformed into:

```
meaneven = 0;
```

The transformed slice S_1 is shown in Figure 14. Note that the dependence on variables `evens` and `noevens` (in line 2) has been broken after the above DRT transformations are applied. Applying the DRT algorithm to the code fragment from lines 18 to 24, the transformed slice S_2 shown in Figure 15 is obtained.

Slicing S_2 on the variable `evendifference`, the amorphous slice in Figure 16 is obtained (a `C` version is depicted in in Figure 17). In the amorphous slice the bug manifests itself as the fact that the assignment to `evendifference` is defined in terms of the number of elements in the array, `n`, and that no mention of the variables `evens` and `noevens` remains. This makes the presence of the bug very evident. The general aim

```
1   total=0;  i=0;
2   evens=0;  noevens=0;
3   odds=0;   noodds=0;
4   scanf("%d",&n);
5   while(i<=n){
6     evenflag = A[i] % 2;
7     evenflag=0;
8     if(0)       /* bug */
9       {evens = evens + A[i];
10       noevens = noevens + 1; }
11    else
12      {odds = odds + A[i];
13       noodds = noodds + 1; }
14    total = total + A[i];
15    i = i + 1;
16  }
17  if(noevens!=0) meaneven=evens/noevens;
18  else meaneven = 0;
19  if(noodds!=0) meanodd = odds/noodds;
20  else meanodd = 0;
21  mean = total/(n+1);
22  n = n+1;
23  evendifference = abs(meaneven-mean);
24  odddifference = abs(meanodd-mean);
```

Figure 11: The C Program P_1: Side-effect free version of P_0

of any implementation of amorphous slicing is to remove as much of the unwanted semantics of the program as possible, creating the simplest possible program which retains the effect of the original on the variables of interest. Of course, as with syntax-preserving slicing, minimal slices are not computable. However, any progress in removing unwanted semantics may improve the chances that any faults will be detected.

6 Advantages and Disadvantages of an AST–based Approach

Traditionally, syntax–preserving slicing has been performed using the System Dependence Graph and its variations [8, 34, 48, 53, 54, 67, 68, 82]. Binkley [9] showed how transformation rules could be incorporated into the SDG so that it could be used to construct amorphous slices. A detailed algorithm for amorphous slicing using the SDG is presented by Harman, Binkley and Danicic [35]. This section addresses the advantages and disadvantages of the two approaches: AST-based and SDG-based amorphous slicing. Currently available implementations of the SDG-based algorithm, CodeSurfer [33] and the Aristotle system [57] target the C programming language, so examples in this section will be presented in the C language.

Broadly speaking, the advantages of the SDG-based approach over the AST–based approach are:

- control and data dependence information are locally available to each node,
- computational effort involved in some of the preconditions on transformations is reduced and
- the execution order independence of the SDG removes the need for code motion transformations.

However, the disadvantages of the SDG-based approach include:

- the execution order independence of the SDG makes some transformations harder to apply and
- the granularity of the SDG-representation is occasionally too coarse.

```
1   total:=0;  i:=0;
2   evens:=0;  noevens:=0;
3   odds:=0;   noodds:=0;
4   n:=n0;
5   WHILE i <= n DO
6     evenflag := A[i] % 2;
7     evenflag := 0;
8     IF FALSE THEN
9       evens := evens + A[i];
10      noevens := noevens + 1
11    ELSE
12      odds := odds + A[i];
13      noodds := noodds + 1 FI;
14    total := total + A[i];
15    i := i + 1;
16  OD
17  IF noevens<>0 THEN meaneven:=evens/noevens;
18  ELSE meaneven := 0 FI;
19  IF noodds<>0 THEN meanodd := odds/noodds;
20  ELSE meanodd := 0 FI;
21  mean := total/(n+1);
22  n := n+1;
23  evendifference := abs(meaneven-mean);
24  odddifference := abs(meanodd-mean);
```

Figure 12: P_2: WSL program converted from C program P_1

The SDG-based syntax-preserving slicing algorithm [48, 61] has also been extended to handle slicing programs with `goto` statements [4, 14, 52], although there are other non-SDG-based approaches to handling `goto` statements [1, 38]. However, for amorphous slicing, `goto` statements present less of an issue, because they can be transformed out using standard restructuring transformations [65, 73].

Section 6.1 explains the advantages of the SDG–based approach, while Section 6.2 explains the disadvantages. In general, the SDG–based approach can be thought of as a compilation, by contrast with the AST–based approach which is interpretive. The strengths and weaknesses that accrue from the choice between compilation-based strategies and interpretation-based strategies also apply. Section 6.3 explores this observation in more detail.

6.1 Advantages of the SDG–Based Approach over the AST–Based Approach

The advantages of using the SDG come from having the necessary control and data flow information in one place. This has long been known to simplify the computation of syntax-preserving slices [48, 61]. It also assists the computation of the transformations used in amorphous slicing, by reformulating logical constraints in terms of dependence edges. This reformulation makes implementation easier and more efficient.

To illustrate one advantage, consider the statement order information ubiquitous in the AST. This information is not the necessary ordering of components captured by dependence edges in an SDG, but rather the over specification of order present in a program's source code. To deal with statement order, the AST algorithm must employ many transformations to achieve a relatively simple result, which can be achieved in a single transformation using the SDG.

For example, consider the two programs shown below (in which $r \notin \mathbf{REF}(B)$). The AST-based approach requires rules that allow statements to "pass-by" one another. For instance, the Push Assignment rule allows Statement A to pass-by Statement B yielding Program 2 from Program 1. In Program 2, the two assignments to `r` can be combined (by the Unfold Assignment rule) enabling further simplification. There is no need for a

```
1   total:=0;  i:=0;
2   evens:=0;  noevens:=0;
3
4   n:=n0;
5   WHILE i <= n DO
6
7
8     IF FALSE THEN
9       evens := evens + A[i];
10      noevens := noevens + 1 FI;
11
12
13
14    total := total + A[i];
15    i := i + 1;
16  OD
17  IF noevens<>0 THEN meaneven:=evens/noevens;
18  ELSE meaneven := 0 FI;
19
20
21  mean := total/(n+1);
22
23  evendifference := abs(meaneven-mean);
24
```

Figure 13: Conventional slice S_0 w.r.t the variable `evendifference` at the end of program P_1

Push Assignment rule in the SDG, as statement order is not represented. In fact, the SDGs for Program 1 and Program 2 are identical.

	Program 1	Program 2
A:	`r = r && x`	`y = ···`
B:	`y = ···`	`r = r && x`
	`r = r && y`	`r = r && y`

In general, where two statements `x` $= e_1$ and `y` $= e_2$ are separated by n assignments to variables other than `x`, $n+1$ applications of the Push Assignment rule are required to relocate the assignment to `x` to the statement following the assignment to `y`. This is achieved by one application of the Flow-Edge Removal rule of the SDG-based approach [35], defined below:

Rule 10 (Flow Edge Removal)

$$\frac{v \rightarrow_f u \ \wedge \ v : \text{“}a = e\text{”} \ \wedge \ allowable(a, v, u)}{\begin{array}{c} RHS(label(u))[e/a] \\ SDG \leftarrow SDG - \{v \rightarrow_f u\} \ \cup \ \{x \rightarrow_f u \mid x \rightarrow_f v\} \end{array}}$$

The Flow-Edge Removal Rule (10) states how an expression e can be propagated along a flow edge of the SDG from an assignment vertex v labeled “$a = e$” to a vertex u. For this rewriting to be allowed (safe), u must be the target of exactly one flow-dependence edge for variable a. Furthermore, if u represents a vertex that also defines a, then the rewriting rule requires the incoming flow edges of u be a subset of the incoming flow edges of v (excluding the edge $v \rightarrow_f u$). Safety ensures that the rewritten graph can be transformed back into a sequential program.

```
1   total:=0;  i:=0;
2   evens:=0;  noevens:=0;
4   n:=n0
5   WHILE i <= n DO
14    total := total + A[i];
15    i := i + 1;
16  OD
18  meaneven := 0;
21  mean := total/(n0+1);
23  evendifference := abs(meaneven-mean);
```

Figure 14: Transformed slice S_1

```
total:=0;  i:=0;
evens:=0;  noevens:=0;
WHILE i <= n0 DO
  total := total + A[i];
  i := i + 1;
OD
evendifference := abs(-total/(n0+1));
meaneven := 0;
mean := total/(n0+1);
n := n0;
```

Figure 15: Transformed slice S_2

6.2 Advantages of the AST–Based Approach over the SDG–Based Approach

While the SDG–based approach has many advantages, it also has some disadvantages. Two of these are now discussed. The first of these is a 'true disadvantage'. The second reflects the granularity at which current implementations [33, 57] operate and so is a disadvantage, merely of current implementations, rather than of the approach itself. The first disadvantage arises from the complete lack of statement order information in the SDG. In some cases, this lack of information makes it 'too easy' to apply transformation rules. That is, transformation rules could potentially be applied in incorrect situations, unless additional edges are added to the SDG to denote statement execution order and to prevent the application of these transformations. The Flow-Edge Removal rule is an excellent example of this problem.

For example, consider the code in Figure 18. Rewriting the flow dependence edge from Statement 2 to Statement 5 is problematic because it is impossible to transform the transformed graph into a sequential program without introducing temporary variables (or undoing the transformations). The reason for this is that the dependences of the rewritten Statement 5 require it to be after Statement 4 ($4 \rightarrow_f 5$), but before Statement

```
total:=0;  i:=0;
WHILE i <= n0 DO
  total := total + A[i];
  i := i + 1;
OD
evendifference := abs(-total/(n+1));
```

Figure 16: Amorphous slice of program P_0 w.r.t. the variable `evendifference` at the end of P_0

```
total=0;  i=0;
scanf("%d",&n);
while(i<=n){
  total = total + A[i];
  i = i + 1;
}
evendifference = abs(-total/(n+1));
```

Figure 17: C language version of the Amorphous slice in Figure 16

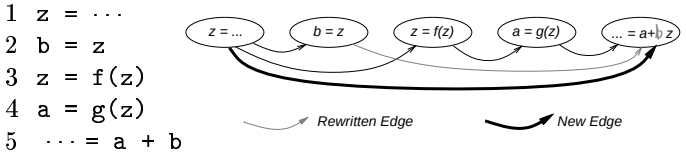


Figure 18: Potential Problems for Flow Edge Removal

3 (where z is redefined). In this example, the dependence on z requires Statement 4 to follow Statement 3 ($3 \rightarrow_f 4$); thus, no satisfactory linear order exists. Note that the resulting SDG is well defined and has the expected semantics.

By contrast, using the AST-based approach, the assignment at Line 2 of this fragment would become 'stuck' were an attempt to be made to 'push' it forward through the remaining three statements. The value assigned to b at line two therefore cannot be pushed to the use of b in Line 5. This problem requires the inclusion of some statement order information in the SDG.

The second disadvantage identified in the current implementation deals with the granularity at which program components are represented in the SDG. CodeSurfer and Aristotle use *statement* level SDGs. That is, for the most part, the SDG has one vertex for each statement in the source program. The main exception to this is the call statement, which includes multiple vertices for each parameter.

At the statement level of granularity certain restrictions are necessary on particular transformations. For example, consider the flow edge from v to u in Figure 19.
At the statement level, removing this flow edge makes it impossible to reconstruct a program from the transformed SDG (without introducing temporary variables) because of the two incoming flow dependence edges for i at vertex u. Thus, this transformation needs to be restricted.

This difficulty can be overcome by moving to a finer granularity for SDG nodes. An 'operator level SDG' includes a vertex for each operator and operand in a statement (as well as the statement itself). The operator-level SDG makes it possible to keep separate multiple distinct definitions of each variable that reaches a vertex. This is illustrated in Figure 20, which shows the *operator-level* SDG for the above code before and after the removal of the flow edge from v to u. The operator level SDG avoids the granularity problem because the two definitions of i reach different operands at u. In this example, it is impossible to transform the resulting graph

Original Code		Rewritten Code	
	i = ···		i = ···
v	x = s[i]	v	x = s[i]
	i = i + 1		i = i + 1
u	x = i + x	u	x = i + s[i]

Figure 19: The Granularity Problem for the SDG

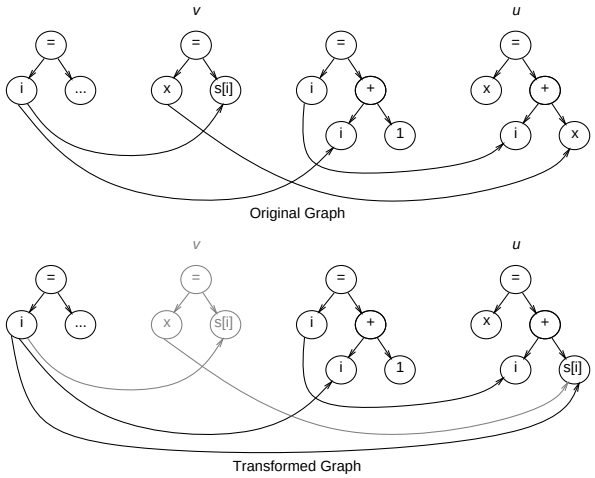


Figure 20: Rewriting at the operator level enables rewritings that are problematic at the statement level.

1	`y := x*x;`	`x1 := a;`
2	`x := y*y;`	`x2 := b;`
3	`y := x*x;`	`x3 := x2 - x1;`
4	`x := y*y;`	`x4 := x3 - x2;`
5	`y := x*x;`	`x5 := x4 - x3;`
...	.....	.....
20	`x := y*y;`	`x20 := x19 - x18;`
...	.....	.....
n	`WHILE f(x,y)>0 DO x := x+1 OD`	$x_n := x_{n-1} - x_{n-2}$
LOC	The best case (to be sliced on x)	The worst case (to be sliced on x_n)

Figure 21: Examples of Best and Worst Case

directly into a program. However, the graph is semantically meaningful and sufficient information exists to reconstruct a program if the introduction of temporary variables is allowed.

6.3 Compilation verses Interpretation

The SDG–approach requires the building of an SDG. This can be computationally expensive. Much of the effort required to build the SDG can be thought of as 'up front analysis effort', which may have to be performed in the AST–based approach at some point when constructing the slice for some chosen criterion. However, some effort in constructing an SDG may be wasted in certain circumstances. For instance a large function or procedure which turns out to be omitted in all slices, will be analyzed when building the SDG, but will not be analysed by the AST–based algorithm.

In a situation where only a single slice is required, the computation effort involved in first building the SDG, and subsequently applying it to a slicing problem may outweigh the computational effort of producing the slice directly from the AST. In situations where many slices are to be computed, the effort required to build the SDG pays off, because this effort is expended only once.

7 Empirical Performance Analysis

The authors implemented LinIAS, a tool which produces amorphous slices for WSL and which is built on top of the FermaT transformation workbench Fermat is available under GPL from

1	x3 := x2 - x1;	y := x*x;
2	x4 := x3 - x2;	x := y*y;
3	y := x*x;	WHILE f(-x1,x,y)>0 DO x := x+1 OD;
4	x := y*y;	y := x*x;
5	WHILE f(x4,x,y)>0 DO x := x+1 OD;	x := y*y;
6	x5 := x4 - x3;	WHILE f(x1-x2,x,y)>0 DO x := x+1 OD
7	x6 := x5 - x4;	
8	y := x*x;	
9	x := y*y;	
10	WHILE f(x6,x,y)>0 DO x := x+1 OD	
LOC	Program	Amorphous slice with respect to x

Figure 22: Synthetic Program Fragment which can be Reduced 40% by Amorphous Slicing

Size (LOC)	Best Case	Worst Case
20	0.01	0.08
30	0.02	0.15
50	0.03	0.38
100	0.11	1.83
200	0.27	8.76
300	0.43	26.70
400	0.63	65.69
500	0.91	148.71

Figure 23: Execution times (in seconds) for a Pentium III, 700MHz

`www.dur.ac.uk/~dcs6mpw/martin/`.

LinIAS is available for use from a web-based interface from:

`http://www.brunel.ac.uk/~cssrllh/slicer/`

The execution time of LinIAS depends predominantly upon the applicability of Push rules and the consequent manipulation of the AST when code motion occurs. This section reports on the performance of dependence reduction with respect to best and worst cases. Performance degrades if more code can be deleted, when the discovery that this code can be deleted requires repeated applications of the iteration embodied in the top level algorithm. Though this is a pathological case, which is unlikely to occur in practice, it allows us to provide an upper bound on the time taken to compute an amorphous slice in terms of the lines of code in the program to be sliced.

An example fragment which denotes the pattern of the worst case is presented in the rightmost column of Figure 21. In this program every assignment is dependent on all previous ones, and each assignment can be pushed down. However, in order to produce the final amorphous slice, the algorithm has to push each line down through the entire program, transforming every statement as it passes and then slicing at the end of each iteration. For this worst case, (human) analysis yields that the value of x_n is one of following:

$$x_n := \begin{cases} (-1)^i a & \text{if } n \text{ is } 3i+1 \\ (-1)^i b & \text{if } n \text{ is } 3i+2 \\ (-1)^i (b-a) & \text{if } n \text{ is } 3i+3 \end{cases}$$

where $i \in \{0, 1, 2, ...\}$.

LinIAS achieves the above result by applying the push rules $n(n-1)/2$ times (for n lines of code). Worst case performance can, therefore, be expected to be quadratic.

In the best case, no lines of code are deleted. This case is 'best' from the point of view of performance, but is clearly less attractive in terms of the results it produces as slicing is only useful when statements are deleted. Fortunately, the best case (in terms of performance) is also a somewhat pathological example, and it is presented merely to provide a lower bound on the computation time required to produce and amorphous slice.

The archetypical code fragment which typifies the best case is presented in the leftmost column of Figure 21. In this program, no dependence can be reduced, the amorphous slice is the original program itself. Although best case performance produces the identity transformation, it involves checking each statement of the program and so best case performance will be linear.

The size of the slice viewed as a 'percentage reduction rate' ranges from 0% in the best case (for performance) to (almost) 100% in the worst case (for performance). In the worst case, the amorphous slice becomes a single line of code and so the simplification produced is dramatic. Combining the patterns of the best case and worst case in Figure 21 we can generate synthetic program fragments which denote other cases in which a precisely defined level of size reduction will be achieved. For example, given 10 as the number of lines of code and a 'slice reduction rate' of 40%, we can construct the example program fragment in Figure 22. This example fragment is a synthetic code fragment in which the algorithm is guaranteed to consider 10 lines of code and produce a 40% reduction in size.

In addition to the time complexity of the approach, there is also a space complexity issue. This is an issue for any transformation–based approach which can lead to expression expansion. In the worst case for space complexity, it is possible to have exponential space complexity in the size of the program. For example[1] consider the program fragment:

x_0 := a;
x_1 := f(x_0, x_0);
x_2 := f(x_1, x_1);
...
x_n := f(x_{n-1}, x_{n-1});

At each application of the PUSH transformation tactic, the expressions are expanded. For example,

x_2 := f(x_1, x_1);

will become (after a single application of PUSH):

x_2 := f(f(x_0, x_0), f(x_0, x_0));

This expansion is clearly exponential, and after a certain point it can be expected that such an exponential expansion would have an impact on execution times. Execution times for this example are shown in Figure 24.

The examples used in the empirical study are synthetic. They have been constructed to explore best and worst case performance of the LinIAS tool. This provides a definite statement about the behaviour of LinIAS with respect to *any* real program.

7.1 Results

The execution times for these best and worst cases for time complexity are shown in Figure 23. Figure 25 shows the execution times for amorphously slicing programs of different sizes to achieve reduction rates 10%, 15%, 30%, 35%, 45% and the worst case respectively. The worst case will be called the '100% case'. Reduction of statements, never quite reaches this 'worst case' value, because the last line always remains. However, as the size of the worst case program to be sliced increases, the size reduction approaches 100%.

For small units, slice construction time is of the order of a few seconds. Currently, these observations remain experimental. Even given the worst case results reported here, there is reason to be confident that optimisations in our algorithm and improvements in processor performance will ensure that these times will tend towards the instantaneous computation of amorphous slices.

[1] This example was suggested by one of this paper's anonymous referees.

Size (LOC)	Worst Case
10	0.06
11	0.13
12	0.24
13	0.45
14	0.91
15	1.81
16	3.69
17	7.21
18	14.84
19	30.21
20	61.57
21	124.21
22	252.62
23	516.66
24	1106.22
25	2230.97
26	4614.62
27	9740.22

Figure 24: Impact of Space Complexity: Execution times (in seconds) for a Pentium III, 700MHz

LinIAS allows a trade-off between precision and speed because it implements an 'anytime' algorithm; it can be terminated at any time within the main loop of the central amorphous slicer (see Figure 4) and the result produced 'so far' will be a valid amorphous slice. This partial result is guaranteed to be a valid amorphous slice, but simply may not be as small as that which could be achieved with a little more patience.

The space complexity results show that the worst case space complexity leads to an exponential increase in time complexity (due to the effort of processing an exponentially increasing expression). The results are shown in Figure 26. After the size of the program increases above about 20 lines of code, the exponential expansion of expressions 'kicks in' with a consequent impact on time complexity. Any such exponential expression expansion will therefore require careful monitoring and action to avoid unacceptable performance implications.

8 Amorphous Conditioned Slicing

Conditioned slicing [13, 19, 25], is a variation of traditional slicing in which the slicing criterion is augmented by a condition. Statements and predicates which cannot affect the values of the variables of interest when the condition is satisfied are removed to form the conditioned slice. A conditioned slice can be thought of as a conditioned *program*, which is subsequently *sliced*. The conditioned program is one which must behave identically to the original only when the condition is satisfied. Conditioning a program can involve simplification, since lines which cannot be executed when the condition is met may be removed.

Traditionally, conditioned slices have been constructed by statement deletion and so they produce syntax-preserving slices. However, there is no reason *not* to combine the conditioning process with amorphous static slicing to yield amorphous conditioned slicing.

A prototype conditioned slicer (called ConSUS) for WSL has been implemented [21] using the same approach to symbolic execution used by the ConSIT tool for conditioned slicing of C programs [19], but based on WSL and based upon the '`Simplify`' transformation of FermaT. The use of FermaT's `Simplify` allows an implementation of a (very light weight, though imprecise) theorem prover. ConSUS is described in more detail elsewhere [21].

For example, consider the UK tax calculation program in Figure 27. The program represents an attempt to capture UK tax regulations concerning the computation of amounts of tax payable, including allowances for a tax payer in the tax year April 1998 to April 1999. Each person has a personal allowance which is an amount of un-taxed income. The personal allowance depends upon the status of the person, reflected by the boolean variables `blind`, `married` and `widowed`, and the integer variable `age`. There are three tax bands, for which tax is charged at the rates of `rate10` (=10%), `rate23` (=23%) and `rate40` (=40%). The width of the 10% tax band

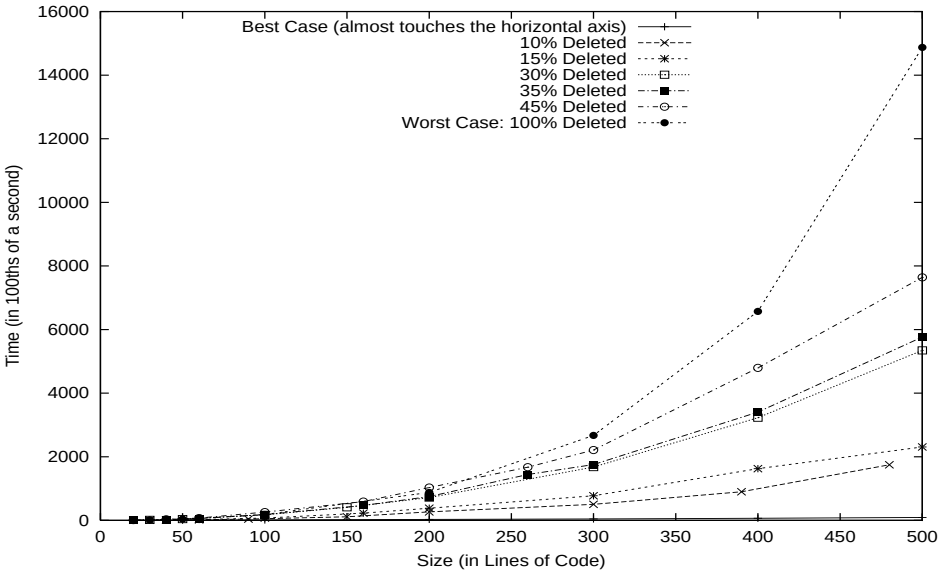


Figure 25: Execution Times for Best and a Set of Worst Cases

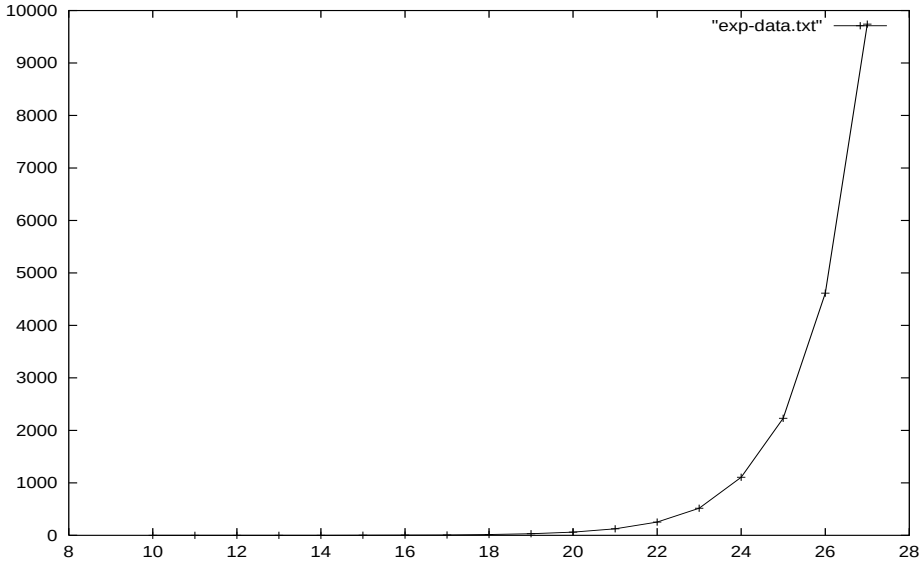


Figure 26: Execution Times for Worst Case Space Complexity

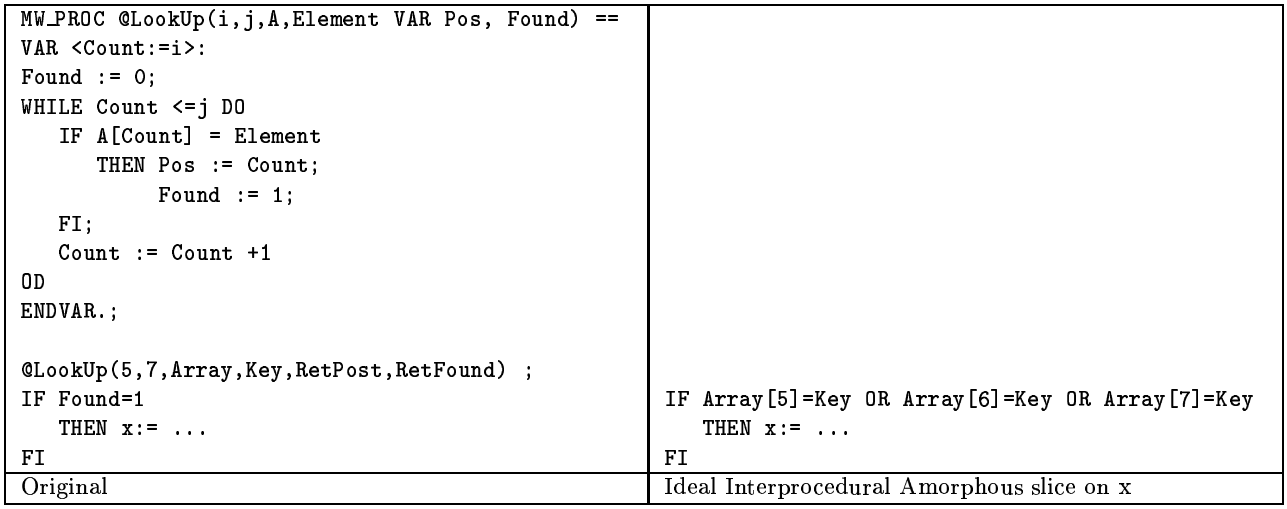
```
IF (age>=75) THEN personal := 5980
ELSE IF (age>=65) THEN personal := 5720
     ELSE personal := 4335 FI FI;

IF (age>=65 AND income >16800) THEN
     IF (4335 > personal-((income-16800) / 2)) THEN personal := 4335
     ELSE personal := personal-((income-16800) / 2) FI FI;

IF (blind =1) THEN personal := personal + 1380 FI;

IF (married=1 AND age >=75) THEN pc10 := 6692
ELSE IF (married=1 AND age >= 65) THEN pc10 := 6625
     ELSE IF (married=1 OR widow=1) THEN pc10 := 3470
          ELSE pc10 := 1500 FI FI FI;

IF (married=1 AND age >= 65 AND income > 16800) THEN
    IF (3470 > pc10-(income-16800) / 2) THEN pc10 := 3470
    ELSE pc10 := pc10-((income-16800) / 2) FI FI;

IF (income - personal <= 0) THEN tax := 0
ELSE income := income - personal;
     IF (income <= pc10) THEN tax := income * rate10
     ELSE tax := pc10 * rate10;
          income := income - pc10;
          IF (income <= 28000) THEN tax := tax + income * rate23
          ELSE tax := tax + 28000 * rate23;
               income := income - 28000 ;
               tax := tax + income * rate40 FI FI FI;
```

Figure 27: UK Income Taxation Calculation Program in WSL

`personal := 4335;`
`pc10 := 3470;`
`income := income - personal;`
`tax := pc10 * rate10;`
`income := income - pc10;`
`tax := tax + 28000 * rate23;`
`income := income - 28000;`
`tax := tax + income * rate40;`
	`tax := 3470*rate10 + 28000*rate23 + (income-35805)*rate40;`

Syntax-Preserving Conditioned Slice	Amorphous Conditioned Slice

Figure 28: The Combination of Syntax-Preserving Conditioned Slicing and Amorphous Slicing



```
MW_PROC @LookUp(i,j,A,Element VAR Pos, Found) ==
VAR <Count:=i>:
Found := 0;
WHILE Count <=j DO
   IF A[Count] = Element
      THEN Pos := Count;
           Found := 1;
   FI;
   Count := Count +1
OD
ENDVAR.;

@LookUp(5,7,Array,Key,RetPost,RetFound) ;
IF Found=1
   THEN x:= ...
FI
```

Original

```
IF Array[5]=Key OR Array[6]=Key OR Array[7]=Key
   THEN x:= ...
FI
```

Ideal Interprocedural Amorphous slice on x

Figure 29: A Desirable Result for Future Work

is subject to the status of the person, while the 23% and 40% are fixed for all individuals.

Given condition {`age>=65 AND age<75 AND income=36000 AND blind=0 AND married=1`}, slicing the program on variable `tax`, using syntax-preserving conditioned slicing, produces the syntax-preserving conditioned slice in leftmost column of Figure 28. However, by replacing the syntax-preserving (static) slicer with the amorphous slicer described in this paper, the (much smaller) amorphous conditioned slice in rightmost column of Figure 28 is obtained. In this way, amorphous slicing may further improve the way in which conditioned slicing helps extract 'business rules' based upon a condition of interest.

9 Related Work

Slicing was introduced by Mark Weiser in 1979 [77] and has been the subject of extensive study since then. In 1984 it was shown that intraprocedural slices could be computed efficiently using the Program Dependence Graph [61]. This result was extended to interprocedural slicing in 1988 [47, 48]. Also in 1988, the first non-static slicing criterion, the dynamic slicing criterion, was introduced [50]. In 1991 the dynamic slicing criterion was extended to the quasi-static criterion [72] and in 1995 dynamic, static and quasi-static criteria were brought within a single generic conditioned [25] (or constrained [29]) criterion, which subsumes static, dynamic and quasi-static criteria [13].

In all these approaches to slicing, the only simplifying transformation used to create slices was statement deletion. This choice was motivated by the original application of slicing to debugging, where the syntax-preserving nature of a slice was important. However, the restriction to statement deletion is not necessary for many of the applications of slicing which have emerged more recently. As greater simplification power is always possible when the restriction to statement deletion is lifted, it is clearly advantageous to lift the restriction in situations where syntax preservation is unimportant.

Several authors have indicated that the syntactic subset requirement of syntax-preserving slicing has been a hindrance to the computation of small slices [14, 56]. Indeed, in his thesis, Weiser immediately recognized and acknowledged ([77], page 6) that it would not always be possible for a slice to be constructed as a purely faithful subset of the original program's syntax.

Many other authors have suggested ways of combining slicing and transformation for a variety of applications including refining the precision of syntax-preserving slicing[27, 70], assisting testing [36], identifying unobservable components in optimising task scheduling [30], register-allocation optimisation [60], partial evaluation [23], restructuring Cobol [28], parallelization [55] and model checking [17].

Amorphous slicing was first introduced by Harman and Danicic [37], and has been developed by Binkley [9] and Harman, Binkley and Danicic [35] and by Ward [76]. Binkley's approach uses the System Dependence

Graph [48], while Ward's approach uses a novel syntax–preserving slicing algorithm,which is currently under development into an augmented system for producing semantic slices [75, 76],which are closely related to amorphous slices. Binkley et al. [11, 35] have shown that amorphous slicing aids program comprehension. Hierons, Harman and Danicic [46] have shown how amorphous slices can be used to (partly) ameliorate the equivalent mutant problem for mutation testing.

10 Future work

The approach described here for dependence reduction across procedure boundaries, was simply to push the intraprocedural analysis into the bodies of procedures and to iterate the slicing and dependence reduction in these bodies. Even this simple-minded approach can produce good results, particularly when combined with conditioning. However, the focus of this paper has been to present the approach and to show that it scales sufficiently well to form a basis for future development.

Work is in progress to extend the algorithm and the LinIAS implementation to exploit constant propagation, loop unfolding and a collection of domain–specific transformations, aimed at producing increased simplification power. The hope is that by combining these more advanced features with our existing dependence reduction approach, good results can be achieved in a variety of specialised domains. For example, consider the program in Figure 29. This program is typified by use of constants in a procedure call, which is often found in machine generated code and code which uses libraries naïvely.

The slicing algorithm defined here used intraprocedural constant propagation. However, it does not propagate constants into procedure calls. Neither does it exploit constant loop bounds, by finite unfolding. For certain applications, heuristic techniques (like loop unfolding) can produce significant results. Future work will involve enhancements to our system to exploit these kinds of transformations to provide more powerful amorphous slicing in domain specific contexts.

11 Summary

This paper has described an interprocedural amorphous slicing system for WSL that mixes dependence reduction transformations and traditional slicing to achieve smaller slices at the interprocedural level. Amorphous slice construction is harder than traditional, syntax-preserving slice construction. The paper shows why this is the case and compares the Abstract Syntax Tree based approach to the more familiar System Dependence Graph based approach. The paper describes an algorithm for dependence reduction transformation, showing how it can be combined with syntax-preserving slicing to produce amorphous slices.

The dependence reduction algorithm has been implemented as part of the LinIAS amorphous slicing system. The paper describes LinIAS and explains how it fits into the overall context of the GUSTT slicing system, which combines amorphous slicing with Side-effect removal, slicing criterion guidance and mechanized formal proofs of the program transformations employed. Empirical results from executions of this implementation show that it scales to applications which work at unit level even in the worst case (which is somewhat pathological and likely to be rare).

12 Acknowledgements

We would like to thank Martin Ward for many helpful and interesting discussions about FermaT and the language WSL, and for providing these tools to the community under GPL. Our work has also benefitted from discussions of slicing and transformation with Ira Baxter, Gerardo Canfora, Jim Cordy, Tom Dean, Mike Ernst, Bogdan Korel, Jens Krinke, Andrea De Lucia, Jurgen Rilling and Hongji Yang.

This paper is an extended version of a paper originally submitted to the 2^{nd} IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002). We would like to thank the referees of the original SCAM version of the paper for their helpful comments and also to thank the referees of the extended, JASE version of the paper for their comments and suggestions.

The GUSTT project is funded by the Engineering and Physical Sciences Research Council, EPSRC under grant reference number GR/M58719. Harman, Danicic and Hu are also supported, in part, by EPSRC grants

GR/R98938/01, GR/M78083 and GR/R43150 and two development grants from DaimlerChrysler AG, Berlin. Binkley is supported in part by National Science Foundation grant CCR-9803665.

References

[1] Hiralal Agrawal. On slicing programs with jump statements. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 302–312, Orlando, Florida, June 20–24 1994. Proceedings in SIGPLAN Notices, 29(6), June 1994.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, techniques and tools.* Addison Wesley, 1986.

[3] Krzsztof R. Apt, Jacob Brunekreef, Vincent Partington, and Andrea Schaerf. Alma-O: An imperative language that supports declarative programming. *ACM Transactions on Programming Languages and Systems*, 20(5):1014–1066, September 1998.

[4] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control–flow. In Peter Fritzson, editor, 1^{st} *Conference on Automated Algorithmic Debugging*, pages 206–222, Linköping, Sweden, 1993. Springer. Also available as University of Wisconsin–Madison, technical report (in extended form), TR-1128, December, 1992.

[5] Keith Bennett, Tim Bull, Eddie Younger, and Z. Luo. Bylands: reverse engineering safety-critical systems. In *IEEE International Conference on Software Maintenance*, pages 358–366. IEEE Computer Society Press, Los Alamitos, California, USA, 1995.

[6] Keith H. Bennett. Do program transformations help reverse engineering? In *IEEE International Conference on Software Maintenance (ICSM'98)*, pages 247–254, Bethesda, Maryland, USA, November 1998. IEEE Computer Society Press, Los Alamitos, California, USA.

[7] T. J. Biggerstaff, B. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *15th International Conference on Software Engineering*, Baltimore, Maryland, May 1993. IEEE Computer Society Press, Los Alamitos, California, USA.

[8] David Wendell Binkley. The application of program slicing to regression testing. In Mark Harman and Keith Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 583–594. Elsevier, 1998.

[9] David Wendell Binkley. Computing amorphous program slices using dependence graphs and a data-flow model. In *ACM Symposium on Applied Computing*, pages 519–525, The Menger, San Antonio, Texas, U.S.A., 1999. ACM Press, New York, NY, USA.

[10] David Wendell Binkley and Keith Brian Gallagher. Program slicing. In Marvin Zelkowitz, editor, *Advances of Computing, Volume 43*, pages 1–50. Academic Press, 1996.

[11] David Wendell Binkley, Mark Harman, L. Ross Raszewski, and Christopher Smith. An empirical study of amorphous slicing as a program comprehension support tool. In 8^{th} *IEEE International Workshop on Program Comprehension (IWPC 2000)*, pages 161–170, Limerick, Ireland, June 2000. IEEE Computer Society Press, Los Alamitos, California, USA.

[12] Tim Bull. *Software maintenance by program transformation in a wide spectrum language.* PhD thesis, University of Durham, UK, School of Engineering and Computer Science, 1994.

[13] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. In Mark Harman and Keith Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 595–607. Elsevier Science B. V., 1998.

[14] Jong–Deok Choi and Jeanne Ferrante. Static slicing in the presence of goto statements. *ACM Transactions on Programming Languages and Systems*, 16(4):1097–1113, July 1994.

[15] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76:96–120, 1988.

[16] Thierry Coquand and Christine Paulin. Inductively defined types (preliminary version). In P. Martin-Löf and G. Mints, editors, *Proceedings Int. Conf. on Computer Logic, COLOG'88, Tallinn, USSR, 12–16 Dec 1988*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer-Verlag, Berlin, 1990.

[17] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from java source code. In 22^{nd} *International Conference on Software Engineering (ICSE'2000)*, pages 439–448. IEEE Computer Society Press, Los Alamitos, California, USA, June 2000.

[18] Cristina Cornes, Judicael Courant, Jean-Christophe Filliatre, Gerard Huet, Pascal Manoury, Cesar Munoz, Chetan Murthy, Christine Paulin-Mohring, Amokrane Saibi, and Benjamin Werner. The coq proof assistant, reference manual, version 5.10. Technical Report RT-0177, Inria, Institut National de Recherche en Informatique et en Automatique, July 1995.

[19] Sebastian Danicic, Chris Fox, Mark Harman, and Rob Mark Hierons. ConSIT: A conditioned program slicer. In *IEEE International Conference on Software Maintenance (ICSM'00)*, pages 216–226, San Jose, California, USA, October 2000. IEEE Computer Society Press, Los Alamitos, California, USA.

[20] Sebastian Danicic and Mark Harman. Espresso: A slicer generator. In *ACM Symposium on Applied Computing, (SAC'00)*, pages 831–839, Como, Italy, March 2000.

[21] Mohammed Daoudi, Sebastian Danicic, John Howroyd, Mark Harman, Chris Fox, and Martin Ward. ConSUS: A scalable approach to conditioned slicing. In *IEEE Working Conference on Reverse Engineering (WCRE 2002)*, Richmond, Virginia, USA, October 2002. IEEE Computer Society Press, Los Alamitos, California, USA. Selected for consideration for the sepcial issue of the Journal of Systems and Software.

[22] John Darlington and Rod M. Burstall. A system which automatically improves programs. *Acta Informatica*, 6:41–60, 1976.

[23] Manuvir Das. *Partial evaluation using dependence graphs.* PhD thesis, University of Wisconsin–Madison, 1998.

[24] Andrea De Lucia. Program slicing: Methods and applications. In 1^{st} *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142–149, Florence, Italy, 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

[25] Andrea De Lucia, Anna Rita Fasolino, and Malcolm Munro. Understanding function behaviours through program slicing. In 4^{th} *IEEE Workshop on Program Comprehension*, pages 9–18, Berlin, Germany, March 1996. IEEE Computer Society Press, Los Alamitos, California, USA.

[26] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, May 1979. An earlier version appeared in *ACM Symposium on Principles of Programming Languages (POPL) , Los Angeles, California* , 1977 pp. 206–214.

[27] Michael D. Ernst. Practical fine–grained static slicing of optimised code. Technical Report MSR-TR-94-14, Microsoft research, Redmond, WA, July 1994.

[28] John Field and C. Ramalingam. Identifying procedural structure in cobol programs. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, volume 24.5 of *Software Engeneering Notes (SEN)*, pages 1–10, N. Y., September 6 1999. ACM Press.

[29] John Field, G. Ramalingam, and Frank Tip. Parametric program slicing. In 22^{nd} *ACM Symposium on Principles of Programming Languages*, pages 379–392, San Francisco, CA, 1995.

[30] Richard Gerber and Seongsoo Hong. Slicing real-time programs for enhanced schedulability. *ACM Transactions on Programming Languages and Systems*, 19(3):525–555, May 1997.

[31] Nicolas E. Gold. *Hypothesis-Based Concept Assignment to Support Software Maintenance.* PhD Thesis, Department of Computer Science, University of Durham, 2000.

[32] Nicolas E. Gold. Hypothesis-based concept assignment to support software maintenance. In *IEEE International Conference on Software Maintenance (ICSM'01)*, pages 545–548, Florence, Italy, November 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

[33] Grammatech Inc. The codesurfer slicing system, 2002.

[34] Rajiv Gupta, Mary Jean Harrold, and Mary Lou Soffa. An approach to regression testing using slicing. In *Proceedings of the IEEE Conference on Software Maintenance*, pages 299–308, Orlando, Florida, USA, 1992. IEEE Computer Society Press, Los Alamitos, California, USA.

[35] Mark Harman, David Wendell Binkley, and Sebastian Danicic. Amorphous program slicing. *Journal of Systems and Software*, 2003. To appear.

[36] Mark Harman and Sebastian Danicic. Using program slicing to simplify testing. *Software Testing, Verification and Reliability*, 5(3):143–162, September 1995.

[37] Mark Harman and Sebastian Danicic. Amorphous program slicing. In 5^{th} *IEEE International Workshop on Program Comprenhesion (IWPC'97)*, pages 70–79, Dearborn, Michigan, USA, May 1997. IEEE Computer Society Press, Los Alamitos, California, USA.

[38] Mark Harman and Sebastian Danicic. A new algorithm for slicing unstructured programs. *Journal of Software Maintenance and Evolution*, 10(6):415–441, 1998.

[39] Mark Harman, Nicolas Gold, Robert Mark Hierons, and David Binkley. Code extraction algorithms which unify slicing and concept assignment. In *IEEE Working Conference on Reverse Engineering (WCRE 2002)*, Richmond, Virginia, USA, October 2002. IEEE Computer Society Press, Los Alamitos, California, USA.

[40] Mark Harman and Robert Mark Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

[41] Mark Harman, Lin Hu, Robert Hierons, André Baresel, and Harmen Sthamer. Improving evolutionary testing by flag removal ('best at GECCO' award winner). In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1359–1366, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[42] Mark Harman, Lin Hu, Robert Mark Hierons, Xingyuan Zhang, Malcolm Munro, José Javier Dolado, Mari Carmen Otero, and Joachim Wegener. A post-placement side-effect removal algorithm. In *IEEE International Conference on Software Maintenance (ICSM 2002)*, pages 2–11, Montreal, Canada, October 2002. IEEE Computer Society Press, Los Alamitos, California, USA.

[43] Mark Harman, Lin Hu, Xingyuan Zhang, and Malcolm Munro. GUSTT: An amorphous slicing system which combines slicing and transformation. In 1^{st} *Workshop on Analysis, Slicing, and Transformation (AST 2001)*, pages 271–280, Stuttgart, October 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

[44] Mark Harman, Lin Hu, Xingyuan Zhang, and Malcolm Munro. Side-effect removal transformation. In 9^{th} *IEEE International Workshop on Program Comprehension (IWPC'01)*, pages 310–319, Toronto, Canada, May 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

[45] Mark Harman, Yoga Sivagurunathan, and Sebastian Danicic. Analysis of dynamic memory access using amorphous slicing. In *IEEE International Conference on Software Maintenance (ICSM'98)*, pages 336–345, Bethesda, Maryland, USA, November 1998. IEEE Computer Society Press, Los Alamitos, California, USA.

[46] Robert Mark Hierons, Mark Harman, and Sebastian Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, 9(4):233–262, 1999.

[47] Susan Horwitz, Thomas Reps, and David Wendell Binkley. Interprocedural slicing using dependence graphs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 25–46, Atlanta, Georgia, June 1988. Proceedings in *SIGPLAN Notices*, 23(7), pp.35–46, 1988.

[48] Susan Horwitz, Thomas Reps, and David Wendell Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[49] V. Karakostas. Intelligent search and acquisition of business knowledge from programs. *Journal of Software Maintenance: Research and Practice*, 4:1–17, 1992.

[50] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988.

[51] Jan Kort, Ralf Lämmel, and Joost Visser. Functional transformation systems. In 9^{th} *International Workshop on Functional and Logic Programming (WFLP'2000)*, Benicassim, Spain, September 2000. Online proceedings at `http://www.dsic.upv.es/~wflp2000/`.

[52] Sumit Kumar and Susan Horwitz. Better slicing of programs with jumps and switches. In *Proceedings of the* 5^{th} *International Conference on Fundamental Approaches to Software Engineering (FASE 2002)*, volume 2306 of *Lecture Notes in Computer Science*, pages 96–112. Springer, 2002.

[53] Loren D. Larsen and Mary Jean Harrold. Slicing object-oriented software. In *Proceedings of the 18th International Conference on Software Engineering*, pages 495–505, Berlin, 1996.

[54] Douglin Liang and Mary Jean Harrold. Slicing objects using the System Dependence Graph. In *IEEE International Conference of Software Maintenance*, pages 358–367, Bethesda, Maryland, USA, November 1998. IEEE Computer Society Press, Los Alamitos, California, USA.

[55] Shih-Wei Liao, Amer Diwan, Robert P. Bosch, Jr., Anwar Ohuloum, and Monica S. Lam. SUIF explorer: An interactive and interprocedural parallelizer. In A. Andrew Chien and Marc Snir, editors, *Proceedings of the 1999 ACM Sigplan Symposium on Principles and Practise of Parallel Programming (PPoPP'99)*, volume 34.8 of *ACM Sigplan Notices*, pages 37–48, A.Y., May 4–6 1999. ACM Press.

[56] James R. Lyle and David Wendell Binkley. Program slicing in the presence of pointers. In *Foundations of Software Engineering*, pages 255–260, Orlando, FL, USA, November 1993.

[57] Mary Jean Harrold et al. The aristotle analysis tool, 2002.

[58] Vadim Maslov. Lazy array data-flow dependence analysis. In ACM, editor, *Conference record of POPL '94, 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages : papers presented at the Symposium : Portland, Oregon, January 17–21, 1994*, pages 311–325, New York, NY 10036, USA, 1994. ACM Press.

[59] Michael Mehlich and Ira Baxter. Mechanical tool support for high integrity software development. In *High Integrity Systems '97*. IEEE Computer Society Press, Los Alamitos, California, USA, 1997.

[60] Cindy Norris and Lori L. Pollock. The design and implementation of RAP: A PDG-based register allocator. *Software Practice and Experience*, 28(4):401–424, April 1998.

[61] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in software development environments. *SIGPLAN Notices*, 19(5):177–184, 1984.

[62] Lahcen Ouarbya, Sebastian Danicic, Dave (Mohammed) Daoudi, Mark Harman, and Chris Fox. A denotational interprocedural program slicer. In *IEEE Working Conference on Reverse Engineering (WCRE 2002)*, Richmond, Virginia, USA, October 2002. IEEE Computer Society Press, Los Alamitos, California, USA.

[63] Helmut A. Partsch. *The Specification and Transformation of Programs: A Formal Approach to Software Development*. Springer, 1990.

[64] Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15(5–6):607–640, 1993.

[65] Lyle Ramshaw. Eliminating goto's while preserving program structure. *Journal of the ACM*, 35(4):893–920, 1988.

[66] Conor Ryan and Paul Walsh. The evolution of provable parallel programs. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 295–302, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[67] Saurabh Sinha and Mary Jean Harrold. Analysis of programs with exception handling constructs. In *International Conference on Software Maintenance*, pages 348–357. IEEE Computer Society Press, Los Alamitos, California, USA, November 1998.

[68] Saurabh Sinha, Mary Jean Harrold, and Gregg Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control-flow. In *Proceedings of the 21st International Conference on Software Engineering*, pages 432–441. ACM Press, May 1999.

[69] Joseph E. Stoy. *Denotational semantics: The Scott–Strachey approach to programming language theory*. MIT Press, 1985. Third edition.

[70] Frank Tip. *Generation of Program Analysis Tools*. PhD thesis, Centrum voor Wiskunde en Informatica, Amsterdam, 1995.

[71] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[72] Guda A. Venkatesh. The semantic approach to program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–28, Toronto, Canada, June 1991. Proceedings in *SIGPLAN Notices*, 26(6), pp.107–119, 1991.

[73] Martin Ward. *Proving Program Refinements and Transformations*. DPhil Thesis, Oxford University, 1989.

[74] Martin Ward. Reverse engineering through formal transformation. *The Computer Journal*, 37(5), 1994.

[75] Martin Ward. The formal approach to source code analysis and manipulation. In 1^{st} *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 185–193, Florence, Italy, 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

[76] Martin Ward. Program slicing via FermaT transformations. In 26^{th} *IEEE Annual Computer Software and Applications Conference (COMPSAC 2002)*, pages 357–362, Oxford, UK, August 2002. IEEE Computer Society Press, Los Alamitos, California, USA.

[77] Mark Weiser. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

[78] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[79] Kenneth Peter Williams. *Evolutionary Algorithms for Automatic Parallelization*. PhD thesis, University of Reading, UK, Department of Computer Science, September 1998.

[80] Xingyuan Zhang, Malcolm Munro, Mark Harman, and Lin Hu. Mechanized operational semantics of WSL. In *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002)*, pages 73–82, Montreal, Canada, October 2002. IEEE Computer Society Press, Los Alamitos, California, USA.

[81] Xingyuan Zhang, Malcolm Munro, Mark Harman, and Lin Hu. Weakest precondition for general recursive programs formalized in Coq. In 15^{th} *International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002)*, pages 332–348, Hampton, Virginia, USA, August 2002. Springer Verlag. LNCS 2410.

[82] Jianjun Zhao. Slicing aspect-oriented software. In 10^{th} *IEEE International Workshop on Program Comprehension*, pages 351–260, Paris, France, June 2002. IEEE Computer Society Press, Los Alamitos, California, USA.